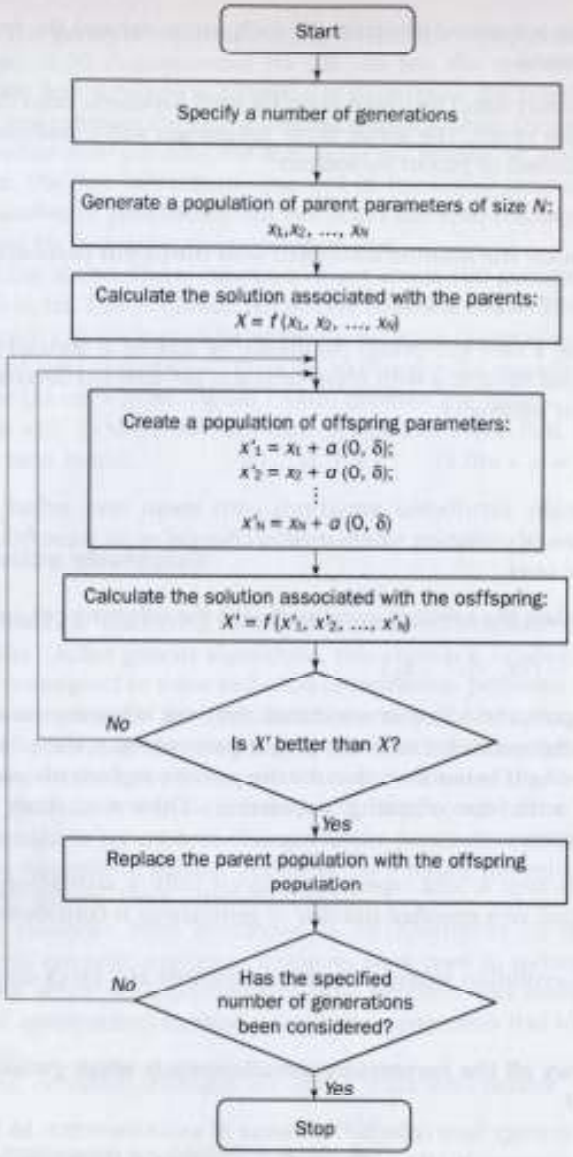

Figure 7.13 Block-diagram of the (1 + 1)-evolution strategy

What are the differences between genetic algorithms and evolution strategies?

The principal difference between a GA and an evolution strategy is that the former uses both crossover and mutation whereas the latter uses only mutation. In addition, when we use an evolution strategy we do not need to represent the problem in a coded form.

Which method works best?

An evolution strategy uses a purely numerical optimisation procedure, similar to a focused Monte Carlo search. GAs are capable of more general applications, but the hardest part of applying a GA is coding the problem. In general, to answer the question as to which method works best, we have to experiment to find out. It is application-dependent.

7.7 Genetic programming

One of the central problems in computer science is how to make computers solve problems without being explicitly programmed to do so. Genetic programming offers a solution through the evolution of computer programs by methods of natural selection. In fact, genetic programming is an extension of the conventional genetic algorithm, but the goal of genetic programming is not just to evolve a bit-string representation of some problem but the computer code that solves the problem. In other words, genetic programming creates computer programs as the solution, while GAs create a string of binary numbers that represent the solution.

Genetic programming is a recent development in the area of evolutionary computation. It was greatly stimulated in the 1990s by John Koza (Koza, 1992, 1994).

How does genetic programming work?

According to Koza, genetic programming searches the space of possible computer programs for a program that is highly fit for solving the problem at hand (Koza, 1992).

Any computer program is a sequence of operations (functions) applied to values (arguments), but different programming languages may include different types of statements and operations, and have different syntactic restrictions. Since genetic programming manipulates programs by applying genetic operators, a programming language should permit a computer program to be manipulated as data and the newly created data to be executed as a program. For these reasons, LISP was chosen as the main language for genetic programming (Koza, 1992).

What is LISP?

LISP, or **List Processor**, is one of the oldest high-level programming languages (FORTRAN is just two years older than LISP). LISP, which was written by John McCarthy in the late 1950s, has become one of the standard languages for artificial intelligence.

LISP has a highly symbol-oriented structure. Its basic data structures are **atoms** and **lists**. An atom is the smallest indivisible element of the LISP syntax. The number *21*, the symbol *X* and the string *'This is a string'* are examples of LISP atoms. A list is an object composed of atoms and/or other lists. LISP lists are

written as an ordered collection of items inside a pair of parentheses. For example, the list

(−(* A B) C)

calls for the application of the subtraction function (−) to two arguments, namely the list (* A B) and the atom C. First, LISP applies the multiplication function (*) to the atoms A and B. Once the list (* A B) is evaluated, LISP applies the subtraction function (−) to the two arguments, and thus evaluates the entire list (−(* A B) C).

Both atoms and lists are called symbolic expressions or **S-expressions**. In LISP, all data and all programs are S-expressions. This gives LISP the ability to operate on programs as if they were data. In other words, LISP programs can modify themselves or even write other LISP programs. This remarkable property of LISP makes it very attractive for genetic programming.

Any LISP S-expression can be depicted as a rooted point-labelled tree with ordered branches. Figure 7.14 shows the tree corresponding to the S-expression (−(* A B) C). This tree has five points, each of which represents either a function or a terminal. The two internal points of the tree are labelled with functions (−) and (*). Note that the root of the tree is the function appearing just inside the leftmost opening parenthesis of the S-expression. The three external points of the tree, also called leaves, are labelled with terminals A, B and C. In the graphical representation, the branches are ordered because the order of the arguments in many functions directly affects the results.

How do we apply genetic programming to a problem?

Before applying genetic programming to a problem, we must accomplish five **preparatory steps** (Koza, 1994):

1. Determine the set of terminals.
2. Select the set of primitive functions.
3. Define the fitness function.
4. Decide on the parameters for controlling the run.
5. Choose the method for designating a result of the run.

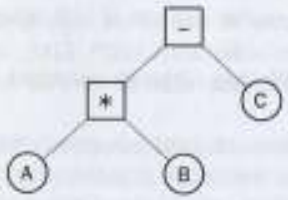

Figure 7.14 Graphical representation of the LISP S-expression (−(* A B) C)

Table 7.3 Ten fitness cases for the Pythagorean Theorem

Side a	Side b	Hypotenuse c	Side a	Side b	Hypotenuse c
3	5	5.830952	12	10	15.620499
8	14	16.124515	21	6	21.840330
18	2	18.110770	7	4	8.062258
32	11	33.837849	16	24	28.844410
4	3	5.000000	2	9	9.219545

The Pythagorean Theorem helps us to illustrate these preparatory steps and demonstrate the potential of genetic programming. The theorem says that the hypotenuse, c, of a right triangle with short sides a and b is given by

$$c = \sqrt{a^2 + b^2}.$$

The aim of genetic programming is to discover a program that matches this function. To measure the performance of the as-yet-undiscovered computer program, we will use a number of different **fitness cases**. The fitness cases for the Pythagorean Theorem are represented by the samples of right triangles in Table 7.3. These fitness cases are chosen at random over a range of values of variables a and b.

Step 1: ***Determine the set of terminals***

The terminals correspond to the inputs of the computer program to be discovered. Our program takes two inputs, a and b.

Step 2: ***Select the set of primitive functions***

The functions can be presented by standard arithmetic operations, standard programming operations, standard mathematical functions, logical functions or domain-specific functions. Our program will use four standard arithmetic operations +, −, * and /, and one mathematical function *sqrt*.

Terminals and primitive functions together constitute the building blocks from which genetic programming constructs a computer program to solve the problem.

Step 3: ***Define the fitness function***

A fitness function evaluates how well a particular computer program can solve the problem. The choice of the fitness function depends on the problem, and may vary greatly from one problem to the next. For our problem, the fitness of the computer program can be measured by the error between the actual result produced by the program and the correct result given by the fitness case. Typically, the error is not measured over just one fitness case, but instead calculated as a sum of the absolute errors over a number of fitness cases. The closer this sum is to zero, the better the computer program.

Step 4: ***Decide on the parameters for controlling the run***
For controlling a run, genetic programming uses the same primary parameters as those used for GAs. They include the population size and the maximum number of generations to be run.

Step 5: ***Choose the method for designating a result of the run***
It is common practice in genetic programming to designate the best-so-far generated program as the result of a run.

Once these five steps are complete, a run can be made. The run of genetic programming starts with a random generation of an initial population of computer programs. Each program is composed of functions $+$, $-$, $*$, $/$ and *sqrt*, and terminals a and b.

In the initial population, all computer programs usually have poor fitness, but some individuals are more fit than others. Just as a fitter chromosome is more likely to be selected for reproduction, so a fitter computer program is more likely to survive by copying itself into the next generation.

Is the crossover operator capable of operating on computer programs?
In genetic programming, the crossover operator operates on two computer programs which are selected on the basis of their fitness. These programs can have different sizes and shapes. The two offspring programs are composed by recombining randomly chosen parts of their parents. For example, consider the following two LISP S-expressions:

$$(/\ (-\ (sqrt\ (+\ (*\ a\ a)\ (-\ a\ b)))\ a)\ (*\ a\ b)),$$

which is equivalent to

$$\frac{\sqrt{a^2+(a-b)}-a}{ab},$$

and

$$(+\ (-\ (sqrt\ (-\ (*\ b\ b)\ a))\ b)\ (sqrt\ (/\ a\ b))),$$

which is equivalent to

$$\left(\sqrt{b^2-a}-b\right)+\sqrt{\frac{a}{b}}.$$

These two S-expressions can be presented as rooted, point-labelled trees with ordered branches as shown in Figure 7.15(a). Internal points of the trees correspond to functions and external points correspond to terminals.

Any point, internal or external, can be chosen as a crossover point. Suppose that the crossover point for the first parent is the function $(*)$, and the crossover

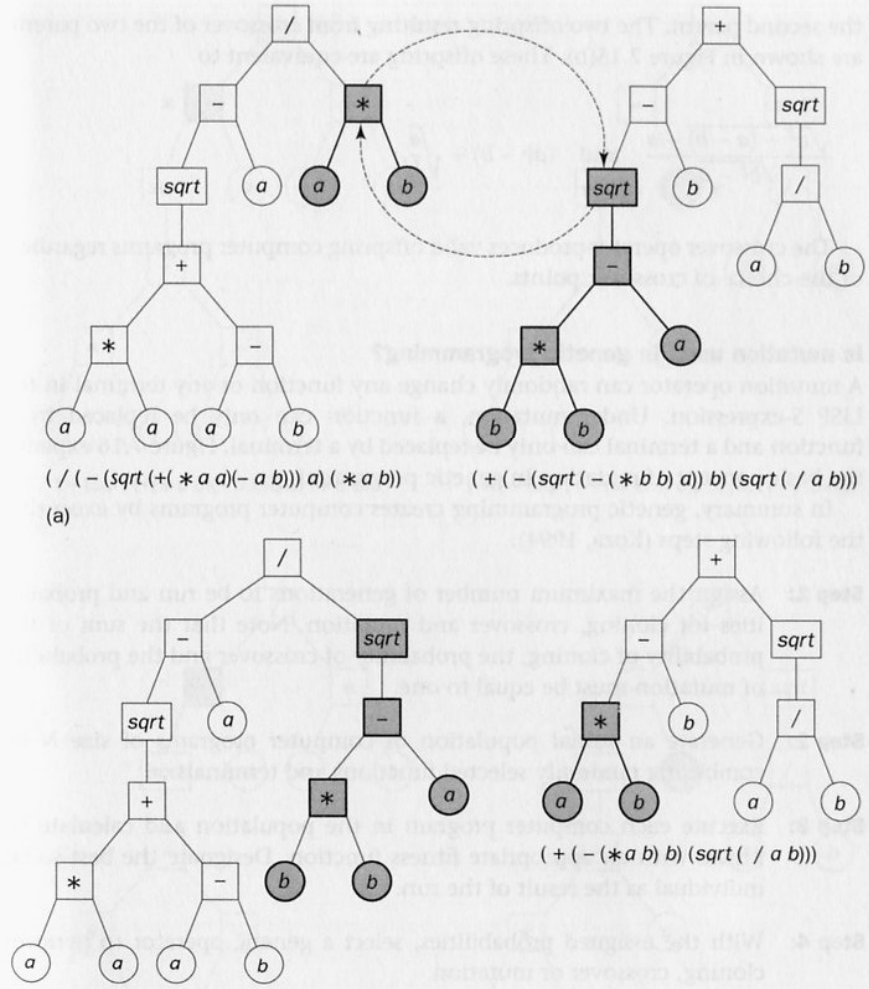

Figure 7.15 Crossover in genetic programming: (a) two parental S-expressions; (b) two offspring S-expressions

point for the second parent is the function *sqrt*. As a result, we obtain the two **crossover fragments** rooted at the chosen crossover points as shown in Figure 7.15(a). The crossover operator creates two offspring by exchanging the crossover fragments of two parents. Thus, the first offspring is created by inserting the crossover fragment of the second parent into the place of the crossover fragment of the first parent. Similarly, the second offspring is created by inserting the crossover fragment of the first parent into the place of the crossover fragment of

the second parent. The two offspring resulting from crossover of the two parents are shown in Figure 7.15(b). These offspring are equivalent to

$$\frac{\sqrt{a^2 + (a - b)} - a}{\sqrt{b^2 - a}} \quad \text{and} \quad (ab - b) + \sqrt{\frac{a}{b}}$$

The crossover operator produces valid offspring computer programs regardless of the choice of crossover points.

Is mutation used in genetic programming?

A mutation operator can randomly change any function or any terminal in the LISP S-expression. Under mutation, a function can only be replaced by a function and a terminal can only be replaced by a terminal. Figure 7.16 explains the basic concept of mutation in genetic programming.

In summary, genetic programming creates computer programs by executing the following steps (Koza, 1994):

Step 1: Assign the maximum number of generations to be run and probabilities for cloning, crossover and mutation. Note that the sum of the probability of cloning, the probability of crossover and the probability of mutation must be equal to one.

Step 2: Generate an initial population of computer programs of size N by combining randomly selected functions and terminals.

Step 3: Execute each computer program in the population and calculate its fitness with an appropriate fitness function. Designate the best-so-far individual as the result of the run.

Step 4: With the assigned probabilities, select a genetic operator to perform cloning, crossover or mutation.

Step 5: If the cloning operator is chosen, select one computer program from the current population of programs and copy it into a new population.

If the crossover operator is chosen, select a pair of computer programs from the current population, create a pair of offspring programs and place them into the new population.

If the mutation operator is chosen, select one computer program from the current population, perform mutation and place the mutant into the new population.

All programs are selected with a probability based on their fitness (i.e., the higher the fitness, the more likely the program is to be selected).

Step 6: Repeat Step 4 until the size of the new population of computer programs becomes equal to the size of the initial population, N.

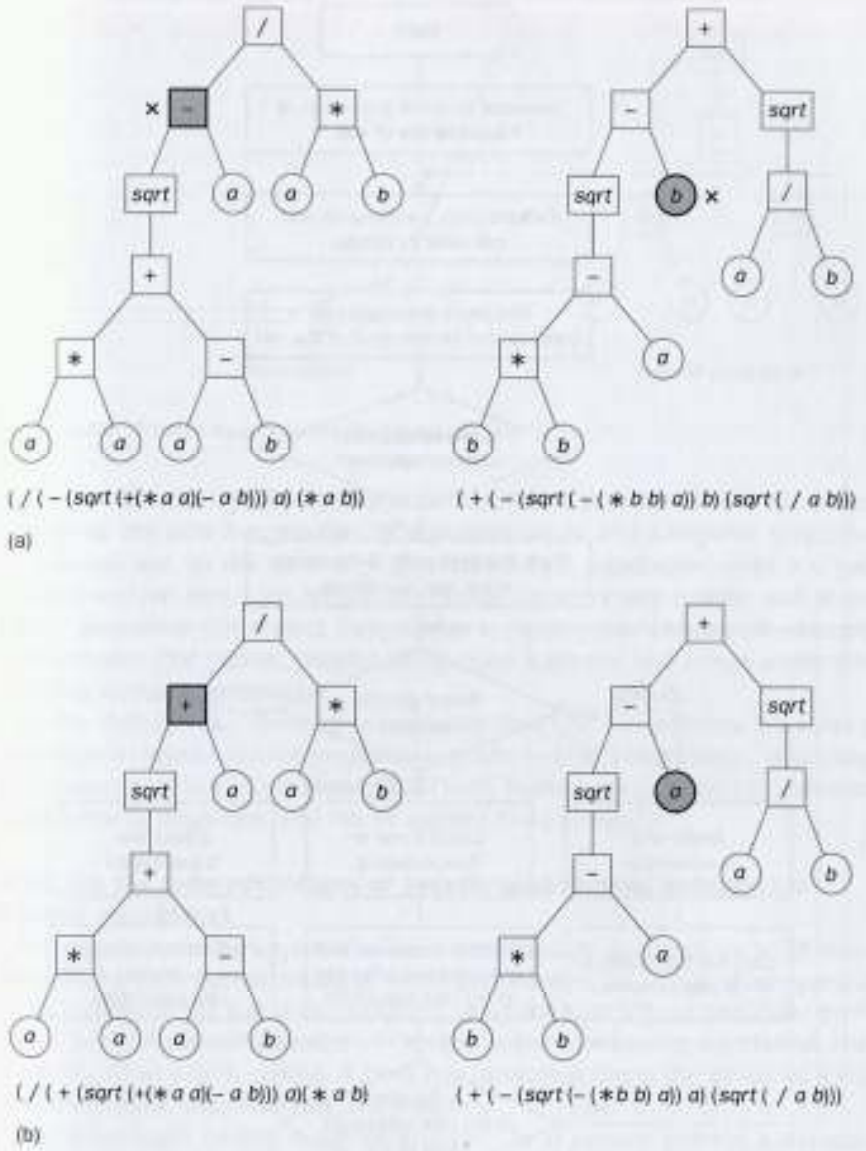

Figure 7.16 Mutation in genetic programming: (a) original S-expressions; (b) mutated S-expressions

Step 7: Replace the current (parent) population with the new (offspring) population.

Step 8: Go to Step 3 and repeat the process until the termination criterion is satisfied.

Figure 7.17 is a flowchart representing the above steps of genetic programming.

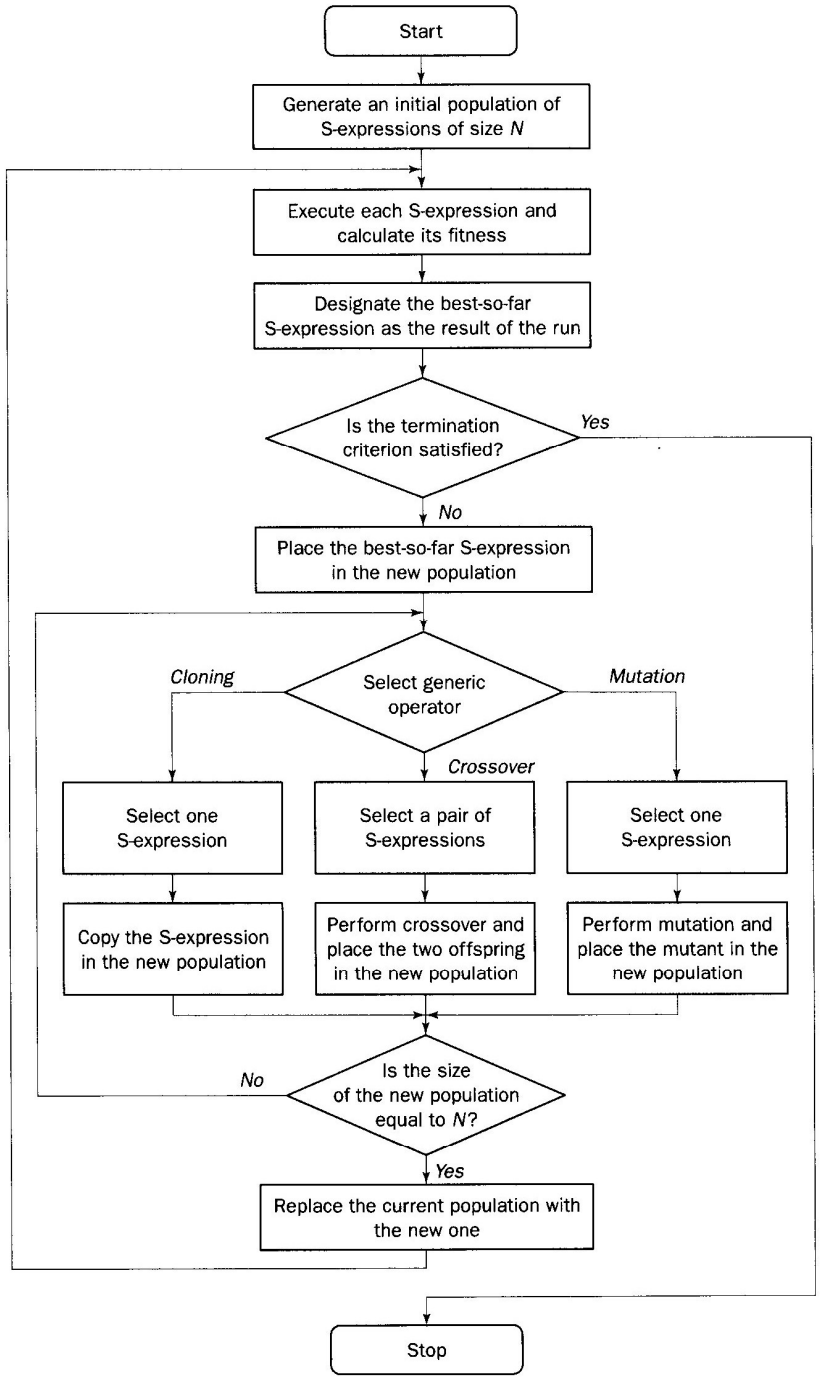

Figure 7.17 Flowchart for genetic programming

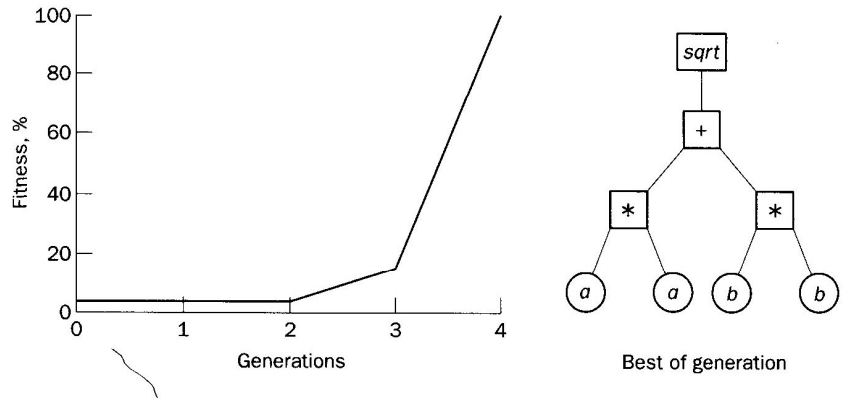

Figure 7.18 Fitness history of the best S-expression

Let us now return to the Pythagorean Theorem. Figure 7.18 shows the fitness history of the best S-expression in a population of 500 computer programs. As you can see, in the randomly generated initial population, even the best S-expression has very poor fitness. But fitness improves very rapidly, and at the fourth generation the correct S-expression is reproduced. This simple example demonstrates that genetic programming offers a general and robust method of evolving computer programs.

In the Pythagorean Theorem example, we used LISP S-expressions but there is no reason to restrict genetic programming only to LISP S-expressions. It can also be implemented in C, C++, Pascal, FORTRAN, Mathematica, Smalltalk and other programming languages, and can be applied more generally.

What are the main advantages of genetic programming compared to genetic algorithms?

Genetic programming applies the same evolutionary approach as a GA does. However, genetic programming is no longer breeding bit strings that represent coded solutions but complete computer programs that solve a particular problem. The fundamental difficulty of GAs lies in the problem representation, that is, in the fixed-length coding. A poor representation limits the power of a GA, and even worse, may lead to a false solution.

A fixed-length coding is rather artificial. As it cannot provide a dynamic variability in length, such a coding often causes considerable redundancy and reduces the efficiency of genetic search. In contrast, genetic programming uses high-level building blocks of variable length. Their size and complexity can change during breeding. Genetic programming works well in a large number of different cases (Koza, 1994) and has many potential applications.

Are there any difficulties?

Despite many successful applications, there is still no proof that genetic programming will scale up to more complex problems that require larger computer programs. And even if it scales up, extensive computer run times may be needed.